Gradual Typing of Erlang Programs: A Wrangler Experience

Konstantinos Sagonas   Daniel Luna

School of Electrical and Computer Engineering, National Technical University of Athens, Greece
Department of Information Technology, Uppsala University, Sweden
kostis@cs.ntua.gr   daniel.luna@it.uu.se

Abstract

Currently most Erlang programs contain no or very little type information. This sometimes makes them unreliable, hard to use, and difficult to understand and maintain. In this paper we describe our experiences from using static analysis tools to gradually add type information to a medium sized Erlang application that we did not write ourselves: the code base of Wrangler. We carefully document the approach we followed, the exact steps we took, and discuss possible difficulties that one is expected to deal with and the effort which is required in the process. We also show the type of software defects that are typically brought forward, the opportunities for code refactoring and improvement, and the expected benefits from embarking in such a project. We have chosen Wrangler for our experiment because the process is better explained on a code base which is small enough so that the interested reader can retrace its steps, yet large enough to make the experiment quite challenging and the experiences worth writing about. However, we have also done something similar on large parts of Erlang/OTP. The result can partly be seen in the source code of Erlang/OTP R12B-3.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Programming by contract; F.3.3 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Documentation, Languages, Reliability

Keywords Erlang, software defect detection, contracts, Dialyzer

1. Introduction

Almost all Erlang applications have so far been written without type information being explicitly present in their code. Of course, this is hardly surprising. After all, Erlang is a dynamically typed language where type information is only implicit during program development. Program testing typically uncovers many typos and type errors and these are corrected in the process. In many cases, type information in the form of (Edoc) comments is added in programs in order to document the intended interfaces of key functions and modules which are part of the API.

In our experience, this mode of developing Erlang programs is far from ideal. Even after extensive testing, many typos and type errors remain in the code. Often these errors appear in the not so commonly executed paths such as those handling serious error situations. Also, type information in the form of comments is often unreliable as it is not checked regularly by the compiler. Such documentation sooner or later is bound to suffer from code rot.

For a number of years now we have been trying to ameliorate this situation by developing and releasing tools that support and promote a different mode of program development in Erlang. Namely, one where most typos, type errors, interface abuses and other software defects are identified automatically using whole program static analysis rather than testing, and where type information is automatically added in the program code, becomes a part of the code, is perhaps manually refined by the programmer and is subsequently automatically checked for validity after program modifications. What's interesting in our approach is that all these are achieved *without* imposing any (restrictive) static type system in the language. Instead, programs can be typed *as gradually as desired* and the programmer has total control of the amount of type information that she wishes to expose and publicly document.

During the last year, we have been practicing this approach on a considerably large part of the Erlang/OTP system. Indeed, nowadays the entire code of the Dialyzer and Typer tools, a large part of the code of the High Performance native code compiler for Erlang (HiPE), and many modules of the standard libraries of Erlang/OTP R12B-3 come with explicit type information. The process has uncovered many software defects, identified some dubious interfaces and a significant number of discrepancies between the published documentation and the actual behavior of key functions of the standard libraries. In the code of Erlang/OTP, the whole process has often been slow and painful, partly because one has to worry about maintaining backwards compatibility and partly because it involves a considerable amount of communication with the Erlang/OTP developers. Nevertheless, overall it has been very rewarding and clearly worth its while. The resulting code is cleaner, easier to understand and maintain, more robust, and much better documented.

This paper aims to document in detail the steps of the program development mode we advocate and have been practicing all this time; both on code produced by our group and on code of Erlang/OTP. By doing so, others who are possibly interested in gradually typing existing Erlang applications can explicitly see what's involved in the process. In particular, they can see both the benefits and costs of using our tools as well as many pitfalls that the more "traditional" mode of Erlang code development involves.

We decided to start with a handicap: we do this experiment on code that we did not write ourselves and for that reason possibly not fully grasp. Also, for the experiment to be interesting, we wanted that the code should be of significant size and publicly available so that others can retrace our steps. After looking around at a handful of open source Erlang projects, we opted for the code of Wrangler, a refactoring tool for Erlang [3].

Erlang'08, September 27, 2008, Victoria, BC, Canada.
Copyright © 2008 ACM 978-1-60558-065-4/08/09. . . $5.00.

```
refac_atom_info.erl:715: Guard test length(M::atom()) can never succeed
refac_batch_rename_mod.erl:161: The call erlang:exit('error',[1..255,...]) will fail
              since it differs in argument position 1 from the success typing arguments: (pid() | port(),any())
refac_util.erl:921: Call to missing or unexported function refac_syntax:class_body/1
refac_util.erl:1322: The call erlang:'and'(bool(),[integer()]) will fail
                     since it differs in argument position 2 from the success typing arguments: (bool(),bool())
```

Figure 1. The main defects of Wrangler 0.1 as identified by Dialyzer

The code of Wrangler has various interesting characteristics with respect to what we want to do. First, it has been developed by researchers who are experts in typed functional programming. For this reason, we expected that Wrangler's code base would be written in a type disciplined manner and would not contain (m)any type errors. Second, we expected that its code base would contain an interesting set of uses of higher order functions — possibly more than in most Erlang code bases out there — and this would be challenging for our tools and approach. Third, the authors of Wrangler have been heavily involved in a project related to testing Erlang programs and have used Wrangler in conjunction with sophisticated testing technology such as QuviQ's QuickCheck tool for Erlang [2]. Finally, the authors of Wrangler are aware of the tools of our group: as they acknowledge in Wrangler's homepage they 'make use some of the ideas from Dialyzer'. In short, we expected that this would be a relatively easy task. Let's see what we found.

2. Using Dialyzer on Wrangler

We started our experiment with the first action we recommend to any Erlang project: use Dialyzer [4]. Dialyzer is a static program analyzer that is really easy to use and is particularly good in identifying software defects which may be hidden in Erlang code, especially in program paths which are not exercised by testing. Indeed, as we will see below, it is quite common that these defects remain unnoticed for a long period of time.

2.1 The first experiment: Dialyzer on Wrangler 0.1

To learn something about Wrangler's evolution, we started by obtaining the first version of Wrangler, which was publicly released on the 25th of January 2007. We executed the following commands:

```
> wget http://www.cs.kent.ac.uk/projects/forse/wrangler/
         distel3.3-wrangler/distel-wrangler-0.1.tar.gz
> tar zxvf distel-wrangler-0.1.tar.gz
> cd distel-wrangler-0.1/wrangler
> wc *.erl
   2229    7247   73088 refac_atom_info.erl
           ... 24 more lines suppressed ...
  34784  137281 1198955 total
```

As we can see from the output of the last command, the main body of the code of Wrangler 0.1 contains a total of 25 modules comprising of about 35,000 lines of code. Out of these modules, many are modified versions of Erlang/OTP modules (of the syntax_tools application, the compiler, and two supporting modules of dialyzer).

We postponed making the Wrangler system because we wanted to shake its code first. Instead, we run Dialyzer v1.8.1 as follows:

```
> dialyzer --src -c *.erl
```

This analyzed all Wrangler modules and generated 67 warnings in less than 2 minutes. About 50 of these warnings concerned the refac_epp module and were warnings of the form 'Function F/A will never be called'. Such warnings are typically side-effects of some failing or contract-violating function calls earlier in the same module which in turn makes calls to these functions unreachable. Indeed, these warnings were produced because Dialyzer also identified two calls to the file:open/2 function which violate both its published documentation at www.erlang.org and the explicit type information which exists for this function in the source code of the file module of Erlang/OTP R12B-3. We manually modified the two offending calls to this function by changing them from the old-fashioned one:

```
file:open(Name, read)
```

which is still allowed for backwards compatibility to the more kosher and documentation-conforming one:

```
file:open(Name, [read])
```

In the process, we performed a similar change to two calls to function file:path_open/3. Doing these changes took about two minutes of our time and reduced the number of Dialyzer warnings to 15. About half of these warnings concern modules refac_compile, refac_sys_core_fold and refac_v3_core which are clones of the corresponding modules of Erlang/OTP with only minor modifications. These warnings are genuine errors that have been fixed in Erlang/OTP R12B. We concentrate on four of the remaining warnings that are specific to the code of Wrangler. These warnings are shown in Figure 1.

The first of them concerns a guard that will never succeed. This typically signifies a genuine bug or is a sign of severe programmer confusion. Indeed, very few Erlang programmers fancy writing guards that always fail. In this case the Dialyzer warning identifies a programming error. The corresponding code is shown in Figure 2. As can be seen, M is an atom and the call to length/1 will always fail in this case. However, since this call occurs in a guard context its failure is silenced and can easily remain undetected by testing.

```
handle_call(Call, DefinedVars, State) ->
  ...
  case is_c_atom(Mod) andalso is_c_atom(Fun) of
    true ->
      M = atom_val(Mod),
      ...
      case {M_Loc, Call_Loc} of
        {{L1, C1}, {L2, C2}} ->
           if (L1 < L2) or
               ((L1==L2) and ((C2-C1) > length(M)))
      ...
```

Figure 2. Portion of the code of refac_atom_info.erl

The second warning identifies a call to the exit function with the wrong arity. The corresponding code checks for an error condition and if the condition is met it wants to exit the Wrangler process most probably with a tagged two tuple where the first element is the atom error. Instead, it constructs the call:

```
exit(error,"Can not infer new module names, ...")
```

This is a particularly nasty bug that is very hard to detect by testing. The problem is that this code will abort execution alright, but will do so with a significantly different message than the programmer

```
expand_files([File|Left], Ext, Acc) ->
  case filelib:is_dir(File) of
    true ->
      ...
    false ->
      case filelib:is_regular(File) and
           filename:extension(File) == Ext of
        true  -> expand_files(Left, Ext, [File|Acc]);
        false -> expand_files(Left, Ext, [File])
      end
  end;
```

Figure 3. Portion of the code of `refac_util.erl`

intended. (The `erlang:exit/2` function throws an exceptions and exits a process in Erlang but expects a different type of term in the first argument and will throw a different exception if called with an atom in the first argument.)

The third warning is simple but quite common in Erlang. The code contains a call to a non-existing function (of an existing module). One does not need Dialyzer to detect this error; the `xref` tool would also have detected it.

The last warning is the most interesting one. The corresponding code is shown in Figure 3. To somebody not very familiar with the idiocyncrancies of the Erlang parser this code looks correct. The problem is that `and` binds stronger than `==` in Erlang and so the `case` expression in the code is parsed as:

```
case (filelib:is_regular(File) and
      filename:extension(File)) == Ext of
```

that is, the code in Figure 3 effectively tries to test a boolean value with the value of `Ext`, instead of being parsed the way that the programmer intended:

```
case filelib:is_regular(File) and
    (filename:extension(File) == Ext) of
```

This bug can be fixed either by adding explicit parentheses as above or by using the `andalso` operator instead of `and`.

Overall, we spent about half and hour understanding and fixing the software defects of Wrangler 0.1 that were identified by Dialyzer. We started from this version of Wrangler because we wanted to see which of Wrangler's defects are long-lived and managed to survive from the first to the current release.

2.2 The second experiment: Dialyzer on Wrangler 0.3

At the time of writing this section (early June 2008), version 0.3 was the most recent snapshot of Wrangler. It was released on the 7th of January 2008, almost a year after version 0.1. The structure of Wrangler's source code has changed a bit and some of the modules of Wrangler 0.1 that were from Erlang/OTP are no longer present. However, many modules of the `syntax_tools` application are still present and some new modules have been added. Including those modules, Wrangler's code consists of 25 modules and about 27,000 lines of code. We run Dialyzer as follows:

```
> cd distel-wrangler-0.3/wrangler/erl
> dialyzer --src -I ../hrl -c *.erl
```

After about 50 seconds, Dialyzer produced warnings many of which were in file `refac_epp` and were due to using an atom rather than a list for the options argument of calls to functions of the `file` module. After manually fixing this issue, about 20 warnings remained.

Some of these warnings were due to confusing one library function with another one and abusing its interface. The `lists` module provides a `concat/1` function. Its published documentation at `www.erlang.org` reads:

```
concat(Things) -> string()

  Types:
    Things = [Thing]
    Thing = atom() | integer() | float() | string()

  Concatenates the text representation of the elements
  of Things.  The elements of Things can be atoms,
  integers, floats or strings.
```

However, the current implementation of the `concat/1` function is more liberal than its documentation claims it is. For example, its implementation in Erlang/OTP R12B-3 allows calls where each `Thing` is a tuple:

```
Eshell V5.6.3  (abort with ^G)
1> lists:concat([[{a,1},{b,2}],[{c,3}]]).
[{a,1},{b,2},{c,3}]
```

Note that the result in this case is not a string. The code of Wrangler is relying on an undocumented behaviour of a library function.

```
%% concat(L) concatenate the list representation of
%%  the elements in L - the elements in L can be atoms,
%%  numbers or strings.  Returns a list of characters.

-type concat_thing() ::
        atom() | integer() | float() | string().
-spec concat([concat_thing()]) -> string().

concat(List) ->
    flatmap(fun thing_to_list/1, List).

thing_to_list(X) when is_integer(X) ->
    ...
```

Figure 4. `lists:concat/1` function annotated with a contract

Misunderstanding or abusing the interface of some library function is a very common software defect in dynamically typed languages such as Erlang. We consider this problem quite severe because an application might give the impression of working alright but this remains so *only* until the library has the same observable undocumented behavior. Of course, this is something that is not guaranteed by the library developers. We have noticed this phenomenon happening again and again — even in our own code! — in Erlang applications. For this reason, we have designed and proposed a *contract language* for Erlang [1] and have already annotated key libraries of Erlang/OTP with their documented interface. Indeed, in Erlang/OTP R12B, the corresponding code in the `lists` module reads as shown in Figure 4. Due to the presence of these contracts, Dialyzer can easily detect such interface abuses and warn the user about them.

In this particular case, the problem is easily fixed. The code of Wrangler can simply use the `lists:append/1` function which has the behaviour that its authors are after. There are 13 calls in total to `lists:concat/1` that should become calls to `lists:append/1`.

After this fix, Dialyzer reports 10 warnings in total. The main ones, those related to Wrangler files not from Erlang/OTP, are shown in Figure 5.

The first and last of them are familiar. They are identical to those in Wrangler 0.1 and have remained unaffected by code evolution and undetected by testing and uses of Wrangler. As mentioned, it is not very surprising that the first of them has remained undetected since the defect appears in error-detection code which is notoriously hard to exercise.

```
refac_batch_rename_mod.erl:161: The call erlang:exit('error',[1..255,...]) will fail
                 since it differs in argument position 1 from the success typing arguments: (pid() | port(),any())
refac_duplicated_code.erl:441: The pattern {'error', _Reason} can never match the type 'false' | {'value',tuple()}
refac_fold_expression.erl:97: The pattern {'error', 'reason'} can never match the type {'error','none'} | {'ok',_}
refac_move_fun.erl:137: The pattern {'eror', Reason} can never match the type {'error',_}
refac_util.erl:921: Call to missing or unexported function refac_syntax:class_body/1
```

Figure 5. The main defects of Wrangler 0.3 as identified by Dialyzer

```
trim_clones(FileNames, Cs, MinLength, MinClones) ->
  ...
    case lists:keysearch(File1, 1, AnnASTs) of
      {value, {File1, AnnAST}} ->
        ...
      {error, _Reason} -> {false, {Range, Len, F}}
    end
  ...
```

Figure 6. Portion of the code of `refac_duplicated_code.erl`

The second warning is due to confusion about the possible return values of the `lists:keysearch/3` function. The offending code is shown in Figure 6. We have seen similar defects in various other Erlang code bases. The remaining warnings are simple typos in error checking code. Similar defects have a tendency to remain unnoticed for a long time.

We manually corrected these problems but for the last one (the call to the missing function) which we did not know how to fix. The whole process, including referring to Erlang/OTP's documentation and code to verify issues related to `lists:concat/1` vs. `lists:append/1`, took us a bit more than two hours. With an almost warning-free code base, we could start adding contracts to the code of Wrangler in order to robustify its API and in the hope of identifying more defects and interface abuses. Let's see where this got us.

3. Adding Contracts to Wrangler

The second action we recommend to any Erlang application is to expose as much type information about functions and modules as possible and make this information part of the code. Typically, type information is only implicit in most Erlang programs. Making it more explicit can happen in the following two ways:

Add explicit type guards in key places in the code. Such an action has the advantage that it exposes type information to static analysis tools such as Dialyzer and at the same time ensures that calls to these functions will fail if they violate these type tests during program execution. One disadvantage is that there is a runtime cost associated with this action, but this cost is typically quite small. A more serious disadvantage is that programs may not be prepared to gracefully handle such failures.

Add type declarations and contracts. Type declarations can give convenient names to key data structures which can then be used to document function and module interfaces. Such type information can then be used by Dialyzer to detect interface violations without occurring any runtime overhead. Quite often such information already exists in comments: either in Edoc format or even in plain text.

Of course, these two methods of exposing type information are not mutually exclusive and projects can employ the combination that is best suited for each situation in hand.

In the case of Wrangler 0.3, its source code already contains a fair amount of `@spec` annotations (336 in total). However, the bulk of these annotations is in files that are minor modifications of Erlang/OTP modules. Because for the more up-to-date version of some of these modules (the ones in Erlang/OTP R12B-3) we had already performed a similar action to the one we will describe in this section, we decided to focus on the `@spec` annotations in modules that have been written entirely by Wrangler's authors. There are 15 such modules but three of them (`refac_module_graph`, `wrangler_distel` and `wrangler_options`) contain no annotations. In the remaining 12 modules there are 54 `@spec` annotations in total. Their breakdown according to module is shown in Table 1.

module	@specs
refac_batch_rename_mod	1
refac_duplicated_code	1
refac_expr_search	1
refac_fold_expression	2
refac_gen	7
refac_move_fun	2
refac_new_fun	1
refac_rename_fun	2
refac_rename_mod	2
refac_rename_var	3
refac_util	21
wrangler	11

Table 1. Number of `@specs` in modules of Wrangler 0.3; modules with no `@specs` and modules from Erlang/OTP have been excluded

3.1 Turning `@spec` annotations into `-spec` declarations

At least syntacticly, converting an existing `@spec` annotation into a `-spec` declaration is a rather straightforward procedure. For example, in `refac_batch_rename_mod.erl` the `@spec` annotation:

```
%% @spec batch_rename_mod(OldNamePattern::string(),
%%                        NewNamePattern::string(),
%%                        SearchPaths::[string()]) ->
%%        ok | {error, string()}
```

can immediately be turned into:

```
    -spec batch_rename_mod(OldNamePattern::string(),
                           NewNamePattern::string(),
                           SearchPaths::[string()]) ->
              'ok' | {'error', string()}.
```

The single quotes around the atoms are not really needed, but we recommend their use so that it is clear to the reader what e.g. is supposed to be the atom `'ok'`, which denotes a singleton type in the language of types, rather the `ok()` type where the programmer has mistakenly forgotten the parentheses.

Quite often, one also needs to make up names for types which are not built-in types. For example, `refac_duplicated_code.erl` contains the following `@spec` annotation:

```
%% @spec duplicated_code(FileName ::filename(),
%%                       MinLines ::integer(),
%%                       MinClones::integer()) -> term().
```

which, after making some educated guess, can be turned into:

```
-type filename() :: string().
-spec duplicated_code(FileName ::filename(),
                      MinLines ::integer(),
                      MinClones::integer()) -> any().
```

If one continues this way, she is quickly faced with a problem. Because `@spec` annotations are not routinely checked by the compiler or any static analysis tool, many of them have suffered from severe code rot and have become inaccurate, outdated, or even completely wrong. For example, to be correct, let alone precise, the above `-spec` declaration should actually read:

```
-type filename() :: string().
-spec duplicated_code(FileNames::[filename()],
                      MinLines ::[byte()],
                      MinClones::[byte()]) -> any().
```

Note that the problem is not in the type declaration that we introduced but in that the original `@spec` annotation that the file contained is not correct.

Out of curiosity, we performed the following experiment. We converted all 54 `@spec` annotations of Wrangler 0.3 to `-spec` declarations and added very loose type declarations for type names which were not documented in the code: we basically mapped most of these types to `any()`. This makes the contracts containing these types as forgiving as possible. We then run Dialyzer on the Wrangler files. Dialyzer reported a total of 164 warnings! Recall that this was on a set of files which were warning-free without any `-spec` declarations. This is not the first time we experienced this behaviour: Edoc annotations need to be treated with caution.

In our experience, the 'convert all `@specs` at once' approach is very crude. The user is simply overwhelmed by the number of warnings that Dialyzer reports. We recommend the following approach instead.

Start from some easy files. Easy files are either those that do not contain many `@spec` annotations or those that depend on only few other modules. This way, one has the chance to run Dialyzer on a *single* module at a time and correct the defects that Dialyzer identifies on a module-local basis. Then continue this way until all modules have been processed. Note that this is not guaranteed to result with a set of files which, when considered together, can be analyzed by Dialyzer without any warnings. If the warnings that are produced are too many, then analyze the modules by considering the strongly connected components that they form, fix warnings in the process, and expand on this set until all modules can be analyzed warning-free.

Fixing warnings of only one or of a small set of modules is usually quite easy. For example, for the `refac_rename_var` module, one gets the following warning from Dialyzer.

```
refac_rename_var.erl:66:
  The call  cond_check(..., ..., NewName1::atom())
  breaks the contract (..., ..., NewName::string())
```

where one can immediately see that there is something wrong in the last argument of this function; either in the call on line 66 or in the contract of the function (i.e., the `-spec` declaration that we added). Finding out which of these two is to blame is a bit more tricky, especially if one is unfamiliar with the code. Quite often though the module has some code part that gives a strong indication about where to assign blame.

We followed the approach we describe above and converted all `@specs` to `-specs` ending up with a set of modules for which Dialyzer gave no warnings when run on a single module at a time. In the process we had to fix a total of ten erroneous `specs` out of the 54 original ones. The 'local' column of Table 2 shows how these are partitioned per module. We then run Dialyzer on the complete set of modules, which resulted in a total of 42 warnings. (In fact, only 17 if one excludes warnings that are quite clearly a side-effect of some other warning.) In any case, 42 is a much more manageable number than 164. Most warnings were due to eight additional `specs` in the code of Wrangler 0.3 being erroneous, which we also corrected. Their modules are indicated in the 'global' column of Table 2. The whole process took about six hours. Of course, it would have taken us less time had we been familiar with Wrangler's code.

		wrong `@specs`	
module	`@specs`	local	global
`refac_batch_rename_mod`	1		
`refac_duplicated_code`	1	1	
`refac_expr_search`	1		
`refac_fold_expression`	2		
`refac_gen`	7		1
`refac_move_fun`	2		
`refac_new_fun`	1	1	
`refac_rename_fun`	2		
`refac_rename_mod`	2		
`refac_rename_var`	3	2	
`refac_util`	21	6	5
`wrangler`	11		2

Table 2. Wrong `@specs` in Wrangler 0.3; blank entries denote 0

3.2 Fixing defects exposed by `-spec` declarations

When `-spec` declarations become part of the code, interesting software defects are exposed by Dialyzer. For example, the Wrangler file `refac_util.erl` contains the following `@spec` annotation:

```
@spec pos_to_var_name(Node::syntaxTree(), Pos::Pos) ->
         {'ok', {atom(), {Pos, Pos}}} | ...
```

To ease exposition, let us drop the variable names for referring to types, introduce a type declaration for what the authors of Wrangler denote as `Pos`, and fix this annotation so that its return type is actually correct. The intended specification for function `refac_util:pos_to_var_name/2` should read:

```
-type pos() :: {integer(), integer()}.
-spec pos_to_var_name(Node::syntaxTree(), Pos::pos()) ->
         {'ok', {atom(), {pos(), pos()}, cat()}} | ...
```

where `cat()` is some type. In `refac_rename_var.erl` this function is used as shown in Figure 7. In this code, Dialyzer warns that the equality test between `DefinePos`, which is a two tuple, and a singleton list will always fail. Once again, this is a very difficult bug to spot or discover by testing because it is in code which handles exceptional cases. (Under typical executions, the code goes to the `true` branch anyway.)

```
rename_var(Fname, Line, Col, NewName, SearchPaths) ->
  ...
  case refac_util:pos_to_var_name(AST, {Line,Col}) of
    {ok, {VarName, {_, DefinePos}, C}} ->
      if DefinePos == [{0,0}] ->
           {error, "Renaming of ... is not supported!"};
         true ->
           ...  % code that renames the variable here
           case cond_check(AST1, DefinePos, NewName) of
             ...
```

Figure 7. Portion of the code of `refac_rename_var.erl`

Once this problem gets exposed, Dialyzer also warns about other problems further down in the code. Figure 8 shows a small portion of the code of the `cond_check/3` function. The call to `lists:any/2` demands that `Pos`, which comes from `DefinePos`

```
cond_check(Tree, Pos, NewName) ->
  ...
  BdVars = lists:map(fun(_, B, _) -> B end, ...),
  Clash = lists:any(fun(bound, Bds) ->
                       ...
                       F_Member = fun (P) -> ... end,
                       lists:any(F_Member, Pos) and ...
                     end, BdVars),
  ...
```

Figure 8. Portion of the code of `refac_rename_var.erl`

in Figure 7, is a list. This code will surely fail if ever executed. We could not decipher what exactly this `lists:any/2` call and two similar occurrences further down in the code of `cond_check/3` try to do, so we did the best action we could think of: we simply wrapped the `Pos` variables in a list. This silenced all but one Dialyzer warnings on the complete set of files of Wrangler 0.3.

3.3 Strengthening and factoring `-type` declarations

Since we were unfamiliar with Wrangler's code, when adding contracts we initially mapped most types mentioned in `@spec` annotations (like for example the types `syntaxTree()` and `cat()` in the example of the previous section) to the type `any()`. This is the most general type of the type system, representing the set of all Erlang terms. Mapping these types to `any()` has the property that Dialyzer will not report any contract violations due to a mistake in the definitions of these types. On the other hand, it is clear that in most cases these type names denote only a subset of all Erlang terms and mapping them to `any()` is a gross overapproximation. We can and should do better than that.

However, unless one is pretty certain about the values of types, we recommend that initially one is not overly zealous in constraining them. The reason is that over-constrained type declarations can result in a lot of warnings from Dialyzer. As a result, it might be quite hard to find the culprits and correct these warnings in conjunction with erroneous `-spec` declarations. We instead recommend that one first tries to come to a state where the existing `-spec` declarations do not result in any warnings from Dialyzer and only then start constraining the types. Indeed, this is the approach we followed when typing Wrangler.

Sometimes, Edoc `@type` annotations already exist in the files and these can be changed to the corresponding `-type` declarations. Some other times, type declarations are pretty obvious, as e.g. for the case of the `filename()` type that we mapped to `string()`. Finally, often information about types exists in comments or types are pretty clear from the structure of terms and the names of variables. This is for example what we did for `pos()`. In various parts of the code, it was mentioned that this type denotes a pair of integers. Thus, we initially added the declaration:

```
  -type pos() :: {integer(), integer()}.
```

and corrected the warnings reported by Dialyzer. None of them was related to this declaration. Then, looking deeper in the code, we realized that `pos()` denotes the line and column numbers of a position in the program source; the position `{0,0}` was used to denote the default position or the absense of position information. We subsequently refined its declaration to exclude negative integers:

```
  -type pos() :: {non_neg_integer(), non_neg_integer()}.
```

For safety, a Wrangler programmer might want to further constrain this type to appropriate integer ranges for lines and columns that a source file might contain. For example, the above declaration can be refined to:

```
  -type pos() :: {0..100000, 0..200}.
```

In short: like applications, types can be *gradually* refined and strengthened up to the point that the programmer wishes to expose information about sets of values and impose constraints on their uses. This way, programs can protect themselves from accidentally violating these constraints.

Once types are declared, often one notices that the same type definition appears in more than one file. For example, the above type declaration for `pos()` was added and refined in a total of four Wrangler files. It is of course bad software engineering practice to have the same information in different places in the code. One can either place this type definition in a common header file which can then be included by all files that need it, or place it in only one file, say `m.erl`, and then in all other files can use the notation `m:t()` to refer to this `t()` type definition that module `m` contains. For Wrangler, since a `wrangler.hrl` file already existed, we opted for factoring all type declarations that were used in more than one module to this header file.

3.4 Strengthening underspecified `-spec` declarations

The next step is to gradually strengthen some `-spec` declarations, because quite often many of them are underspecified. For example, in the code of Wrangler about a third of all `@spec` annotations specify a return type of `term()` for the corresponding functions. Obviously, this return type is not very precise; most of these functions return terms with a statically known structure.

Luckily, when `specs` become part of the code, there is an easy automatic way to discover the underspecified ones among them:

```
> dialyzer -Wunderspecs --src -I ../hrl -c *.erl
```

Running this command revealed a total of 19 underspecified `-spec` declarations (out of the 54 ones). This was after we strengthened the `-type` declarations; the number would have been 24 if we had not done so.

Correcting the underspecified declarations is quite easy. For example, for one of them Dialyzer reports:

```
refac_duplicated_code.erl:53:
  Type specification for duplicated_code/3 ::
    ([filename()],[byte()],[byte()]) -> any()
  is a supertype of the success typing:
    ([string()],[byte()],[byte()]) -> {'ok',[1..255,...]}
```

and of course it is a simple matter to change the return type in the `-spec` declaration of this function from `any()` to either the return type which is reported by Dialyzer (denoting a two tuple where the second element is a non-empty string) or to the slightly underspecified but much more readable type `{'ok',string()}`.

It is important to note that the success typing information reported by the `-Wunderspecs` option of Dialyzer is a conservative approximation of the behaviour of the function which is safe to use and can be copied and pasted in the file as is. Its use will never result in any additional Dialyzer warnings. Dialyzer does not really need its presence because it is the one that it infers. But there is a good reason to explicitly add this information in the file: it provides useful documentation and from that point on its consistency with the code can be statically checked by Dialyzer.

Sometimes this search for underspecified contracts uncovers repeated patterns which are so common that they deserve their own type declaration. For example, the Dialyzer call above revealed that many Wrangler files define an auxiliary function `application_info/1` that returns a two tuple of the form `{{_,_}, non_neg_integer()}`. Turns out that the two underscores are always atoms and the non-negative integer represents the arity of a function. We thus added the following type declaration:

```
  -type appl_info() :: {{atom(),atom()}, arity()}.
```

```
> erlc +warn_missing_spec -I ../hrl refac_rename_var.erl
./refac_rename_var.erl:166: Warning: missing specification for function pre_cond_check/4
```

```
> typer --show-exported -I ../hrl refac_rename_var.erl
 Unknown functions: [{refac_syntax,get_ann,1}, ...,
                     {refac_util,envs_bounds_frees,1}, ..., {refac_util,write_refactored_files,1}]
%% File: "refac_rename_var.erl"
%% ---------------------------
-spec pre_cond_check(tuple(),_,_,atom()) -> bool().
-spec rename(Tree::syntaxTree(),DefinePos::pos(),NewName::atom()) -> {syntaxTree(),bool()}.
-spec rename_var(FileName::filename(),...,SearchPaths::[string()]) -> {'ok',string()} | {'error',string()}.
```

```
> typer --show-exported -I ../hrl refac_rename_var.erl -T refac_util.erl
 Unknown functions: [{refac_syntax,get_ann,1}, ...,
                     {refac_util,parse_annotate_file,4},{refac_util,post_refac_check,3}]
%% File: "refac_rename_var.erl"
%% ---------------------------
-spec pre_cond_check(tuple(),non_neg_integer(),non_neg_integer(),atom()) -> bool().
-spec rename(Tree::syntaxTree(),DefinePos::pos(),NewName::atom()) -> {syntaxTree(),bool()}.
-spec rename_var(FileName::filename(),...,SearchPaths::[string()]) -> {'ok',string()} | {'error',string()}.
```

Figure 9. Finding missing contracts for exported functions of module `refac_rename_var` using Typer

in the header file of Wrangler although we could refine the two `atom()` types even further.

With the help of Dialyzer, many underspecified contracts can be strengthened more or less automatically. However, one should be aware that Dialyzer does not report all underspecified contracts. Instead, Dialyzer only reports those `-spec` declarations that are found strictly more general than the corresponding *success typings* that it infers for these functions [5]. If there exists even one argument position in the `-spec` declaration which is more specific than the corresponding success typing, Dialyzer will not report these declarations as underspecified. For this reason, one might want to manually inspect all `-spec` declarations to spot arguments and return values whose types are underspecified. In fact, this is what we did for Wrangler 0.3. After we corrected underspecified contracts which Dialyzer reported, we used `grep` to detect `-spec` declarations with an occurrence of the `term()` or `any()` type and manually corrected these. There were an additionally nine such `-spec` declarations. The whole process described in this subsection took about two hours to complete.

3.5 Adding `-spec` declarations for exported functions

To ease development and maintainability of Erlang applications, we recommend that modules contain `-spec` declarations for all their exported functions. This way, at least their public interface is documented and Dialyzer can detect possible violations. To help detect modules whose public interface is not documented, we introduced a new compiler option in Erlang/OTP R12B-3, called `warn_missing_spec`, which warns about missing `-spec` declarations for all exported functions of a module. We used this option on the files of Wrangler 0.3 which are not from Erlang/OTP. The number of existing and missing `specs` for exported functions for these modules is shown in Table 3. As can be seen, only half of the exported functions have a publicly documented interface.

With the help of this new compiler option and of the Typer tool the missing function specifications can also be generated semi-automatically. For example, Figure 9 shows the three commands we used to find the missing contract of module `refac_rename_var`. The first command uses the new compiler option to see the exported functions without specifications; there is only one of them in this module. Subsequently, Typer is used to generate specifications for all exported functions in this module. For all functions with existing specifications (e.g. functions `rename/3` and `rename_var/5`

	@specs	
module	present	missing
`refac_batch_rename_mod`	1	
`refac_duplicated_code`	1	1
`refac_expr_search`	1	2
`refac_fold_expression`	2	
`refac_gen`	2	4
`refac_module_graph`		1
`refac_move_fun`	2	
`refac_new_fun`	1	
`refac_rename_fun`	1	1
`refac_rename_mod`	1	
`refac_rename_var`	2	1
`refac_util`	21	21
`wrangler`	11	
`wrangler_distel`		13
`wrangler_options`		1

Table 3. Number of existing and missing `specs` for all exported functions of Wrangler 0.3 modules; blank entries denote 0

in this case) Typer is printing them as these appear in the file. But Typer also generates conservative approximations of specifications for the remaining functions. As can be seen, the first attempt to generate such a specification for function `pre_cond_check/4` was only partly successful. The generated specification contains no type information for the second and third argument of the function because Typer also complained that it does not know anything about functions of modules `refac_syntax` and `refac_util` that the `refac_rename_var` module is using. By instructing Typer to *trust* the existing function specifications of file `refac_util.erl` (but recall that this module has specifications for only half of its functions), Typer is able to infer an accurate specification for function `pre_cond_check/4`.

Actually, in this particular case, we happened to be somewhat lucky. Module `refac_util` contained type specifications which are sufficient for Typer to infer a relatively accurate type information for `pre_cond_check/4`. However, often this is not the case. In those situations, we recommend that the user starts from *leaf* modules (i.e., modules which do not call functions from other modules), use Typer to annotate their exported functions with contracts, and continue bottom up in the module dependency graph until all modules are annotated with contracts.

One can even be brave and use the `--annotate` option of Typer, which will automatically insert the generated `-specs` in the source code of the file(s) on which Typer is run.

Of course, one must always keep in mind that the specifications that Typer generates are conservative approximations (in fact, they are *success typings*) and will never contain any constraints that are not present or enforced by the source code of the module. In other words, these automatically generated specifications are correct but possibly imprecise. In most cases, the user needs to refine them manually, both in order to strengthen them and in order to use appropriate type names for their arguments. For example, the occurrence of `tuple()` in the specification of `pre_cond_check/4` denotes a `syntaxTree()`.

4. Contacting the Authors of Wrangler

At this point, instead of proceeding on our own, we decided to get in touch with the authors of Wrangler. We sent them our paper with the information it contains up to this point.

In the beginning of July 2008, the code of Wrangler had been extended and somewhat changed compared with the version of January 2008 that we were looking at, but most of our steps could easily be retraced even in the development version of Wrangler. The Wrangler authors confirmed our findings. They also added `-spec` declarations for most exported functions of Wrangler modules. Unfortunately, they added these specifications in one go and were subsequently confronted with many Dialyzer warnings that they could not figure out their cause. So, they asked for our help. Of course, the culprit was that some of the `-specs` that they added were in conflict with the functions' uses. In other words, the Wrangler authors did not only confirm our findings but also corroborated our opinion that converting all `@spec` annotations into `-spec` declarations in one go is something not recommendable in code bases of significant size.

With our help, the erroneous function specifications which were resulting in warnings from Dialyzer were corrected. There were eight of them in a total of about 150 `-spec` declarations. In the process, some of the specifications written by the authors of Wrangler were tightened and a few more were added by us. The end result was a Wrangler code base which was totally free from Dialyzer warnings, more robust, and with better documentation about its main functions. The Wrangler authors were happier but we were still not fully satisfied...

5. Testing Contracts of Wrangler

What troubled us was the following. Because Dialyzer's analysis is conservative and based on approximations, Dialyzer never reports a code discrepancy if it is not absolutely certain that there is something wrong with the code. In particular, all `-spec` declarations are trusted and are assumed correct unless Dialyzer discovers a clear conflict between their definitions and uses. For functions with no calls, for functions whose calls are with arguments whose types are not precise enough, or in cases where the return value is not involved in any explicit pattern matching, contract violations will not be detected or reported.

For this reason, we have created yet another tool that, given a test suite, dynamically checks the validity of `-spec` declarations in a set of files. This tool is not yet publicly available and its interface is subject to changes so we will only describe its main idea here.

Currently, the tool starts with a set of `.beam` files and a test suite which can be called from some top-level function (e.g. `mytest:run/N`) possibly with some arguments. For all files which have been compiled with `debug_info` on (and thus whose `-specs` are retained in the byte code), it will employ runtime monitoring to check the validity of their contracts and record all violations it detects while the test suite is running. The recording of all contact violations happens using the Erlang error logger and can be saved in a file, if so desired. The contract checker is straightforward to use for code bases with an already existing test suite. The only drawback, albeit a serious one, is that the test suite will run significantly slower. However, because all calls to contract-annotated functions originating from non debug-compiled modules will not be checked, the user can fully control which parts of the code base will be contract checked and the amount of runtime overhead to the test suite.

The authors of Wrangler provided us with a small test suite that we used to test the validity of `-spec` declarations in files that were somehow "touched" by this test suite. These files contained a total of 106 `-specs` out of which 55 were checked at least once; the remaining 51 concerned functions that were not called by the test suite. The contract checker detected a total of six contract violations: two in calls to functions and four cases where functions returned a value of different type than promised.

Two of the contract violations involved functions `get_toks/1` and `concat_toks/1` of the heavily called `refac_util` module. They were both due to an erroneous declaration of the `token()` type by the Wrangler authors. This type was declared as:

```
-type token() :: {'var', pos(), atom()}
               | {'integer', pos(), integer()}
               | {'float', pos(), float()}
               | {'char', pos(), char()}
               | {'string', pos(), string()}
               | {'atom', pos(), atom()}
               | {atom(), pos()}
```

but failed to account for the fact that the lexical analyzer also returns white spaces and comments as tokens. We extended this declaration by including the following two cases:

```
               | {'whitespace', pos(), whitespace()}
               | {'comment', pos(), string()}.
```

and added an appropriate definition for the `whitespace()` type.

The `refac_util` module contained another contract violation. The function `get_bound_vars/1` was declared as:

```
%% @doc Return the bound variables of an AST node.

-spec get_bound_vars(Node::syntaxTree()) -> [atom()].
get_bound_vars(Node) ->
  get_bound_vars_1(refac_syntax:get_ann(Node)).
```

failing to account for the fact that a variable annotation can occasionally be a two tuple containing an atom and a position (e.g. `{'Self',{77,11}}`).

The forth violation concerns function `fold_expression/3` of the `refac_fold_expression` module. Its contract reads:

```
-spec fold_expression(filename(),integer(),integer()) ->
         {'ok', [filename()]} | {'error', string()}
```

but it is clear from the code, shown in Figure 10, that this function returns something different than a list of filenames (strings) when the last argument to the `fold_expression/4` function is `emacs`.

A similar, though not the same violation, concerned the return type of `refac_move_fun:move_fun/6`. Finally, the last violation was detected in the contract of function `refac_gen:generalise/5` whose last argument was erroneously specified as being a `dir()` when in fact it should be `[dir()]` (i.e. a list of directories).

After the corresponding changes, the contract checker reported no violations when running Wrangler's test suite. Of course, this does not mean that Wrangler's contracts were not erroneous anymore. Instead, it just means that contracts which were exercised by the test suite accurately reflect their common uses.

```
fold_expression(FileName, Line, Col) ->
    fold_expression(FileName, Line, Col, emacs).

fold_expression(FileName, Line, Col, Editor) ->
  case refac_util:parse_annotate_file(FileName, true, []) of
    {ok, {AnnAST, _Info}} ->
       ...
       Candidates = search_candidate_exprs(AnnAST, FunName, FunClauseDef),
       case Candidates of
         [] -> {error, "No expressions that are suitable for folding against ..."};
          _ -> Regions = case Editor of
                           emacs ->
                             lists:map(fun({{{StartLine, StartCol}, {EndLine, EndCol}},NewExp}) ->
                                           {StartLine, StartCol, EndLine, EndCol, NewExp, {FunClauseDef, ClauseIndex}}
                                       end, Candidates);
                           eclipse -> Candidates
                         end,
               {ok, Regions}  %% or {ok, FunClauseDef, Regions}? CHECK THIS.
       end;
    {error, Reason} -> {error, Reason}
  ...
```

Figure 10. Portion of the code of `refac_fold_expression.erl`

6. Concluding Remarks

In this paper we described in detail the steps needed to gradually type the code base of an existing Erlang application. We carefully documented the methodology we advocate, the effort that is requires, and the pitfalls that it may involve. In most code bases the process is far from straightforward, but with the help of the static and dynamic analysis tools we have developed it can at least be performed semi-automatically.

In our experience, what we have described for the code base of Wrangler in no way refects on its quality as an application. In fact, it is quite typical for most Erlang applications out there on which we have applied Dialyzer. Type information is not a panacea, but having it as part of the code helps in catching some easy to detect programming errors, documents intended uses of functions and results in code which is easier to understand and whose correctness is easier to maintain.

Acknowledgements

The research of the second author has been supported in part by a grant from the Swedish Research Council (Vetenskapsrådet). We thank Huiqing Li and Simon Thompson for confirming our findings, giving us access to their repository and sending us a test suite for Wrangler.

References

[1] M. Jiménez, T. Lindahl, and K. Sagonas. A language for specifying type contracts in Erlang and its interaction with success typings. In *Proceedings of the 2007 ACM SIGPLAN Erlang Workshop*, pages 11–17, New York, NY, USA, Sept. 2007. ACM Press.

[2] H. Li and S. Thompson. Testing Erlang refactorings with QuickCheck. In *Pre-proceedings of Implementation of Functional Languages*, Sept. 2007.

[3] H. Li and S. Thompson. Tool support for refactoring functional programs. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 199–203. ACM Press, Jan. 2008.

[4] T. Lindahl and K. Sagonas. Detecting software defects in telecom applications through lightweight static analysis: A war story. In C. Wei-Ngan, editor, *Programming Languages and Systems: Proceedings of the Second Asian Symposium (APLAS'04)*, volume 3302 of *LNCS*, pages 91–106. Springer, Nov. 2004.

[5] T. Lindahl and K. Sagonas. Practical type inference based on success typings. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 167–178, New York, NY, USA, 2006. ACM Press.